TU Clausthal

Clausthal University of Technology

Defining Domain Specific Operational Semantics for Activity Diagrams

Christoph Knieke, Björn Schindler, Ursula Goltz and Andreas Rausch

IfI Technical Report Series IfI-12-04

Department of Informatics
Clausthal University of Technology

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Federico Schlesinger
Contact: federico.schlesinger@tu-clausthal.de
URL: http://www.in.tu-clausthal.de/forschung/technical-reports/
ISSN: 1860-8477

The IfI Review Board

Defining Domain Specific Operational Semantics for Activity Diagrams

Christoph Knieke, Björn Schindler, Ursula Goltz and Andreas Rausch

C. Knieke, B. Schindler, A. Rausch at Technische Universität Clausthal, Clausthal-Zellerfeld, Germany, (christop.knieke, bjoern.schindler, andreas.rausch)@tu-clausthal.de
U. Goltz at Technische Universität Braunschweig, Braunschweig, Germany, ursula.goltz@ips.cs.tu-bs.de

Abstract

Since the major revision 2 of the Unified Modeling Language (UML), activity diagrams have acquired many new features, e.g. hierarchy, data flow and signals. Thus, UML 2 activity diagrams are one of the most versatile formalisms, and can be applied in different domains. Activity diagrams are supported by a number of tools enabling for instance the execution of activity models. Based on the domain these tools have specific requirements and need an adequate interpretation of the informal UML semantics. We propose a foundation for a framework which enables composition of operational semantics out of fundamental semantic constructs. These constructs provide options for domain specific variants. As an example, we introduce two different tool developments based on particular operational semantics composed by our approach. One tool focuses on the modeling of information systems whereas the other tool is aimed at the modeling of reactive systems.

1 Introduction

In the recent years, the importance of requirements engineering in large software development projects is increasing significantly. Many approaches for requirements elicitation and specification of software systems are based on the development of formal requirements models instead of informal textual descriptions. One of the most common modeling languages supporting a graphical description of software systems is the Unified Modeling Language (UML) [OMG, 2009]. Processes and system behavior are modeled by behavior diagrams of the UML, e.g. by activity diagrams.

Since the major revision 2 of the UML [OMG, 2009], activity diagrams have acquired many new features, e.g. hierarchy, data flow and signals. Thus,

UML 2 activity diagrams are one of the most versatile formalisms, and can be applied in different domains, e.g. for the modeling of business processes, algorithms, reactive systems, and web applications. In comparison to UML 1.x, activity diagrams of version 2 have a token flow semantics based on Petri nets, which is described by informal text.

Activity diagrams are supported by a number of tools enabling not only the modeling but also the execution of activity models. Thereby, models can be verified and validated by simulation. Tools for such issues have to meet various requirements, depending on the domain of the system under development. For instance, the user of a simulator may validate the model in various ways. In information systems (e.g. big financial systems) the data processed by the system may be observed and various execution paths may be tested step by step. Therefore, the user has to be involved in system decisions. Consequently the system has to stop at control nodes (see Fig. 1). At reactive systems (e.g. control units for robots) the behavior of external systems may be used for validation. The simulation has to realize, for instance, an exact concurrent execution. At the process of Fig. 1 actions A2 and A4 are modeled to be executed concurrently. Hence, connected control nodes may have to be processed at once.

This example shows that tools need an adequate interpretation of the UML activity diagram semantics, which goes beyond the provided options of the UML superstructure. Nevertheless, we observed that these semantics are based on the same semantical constructs. Depending on the domain, variants of these semantical constructs are used. To achieve an understanding of the semantics of activity diagrams independently from the domain, common semantical constructs as well as the variants have to be analyzed. This enables unification and tool support for defining domain specific operational semantics for UML 2 activity diagrams.

We propose a foundation for a framework which enables composition of operational semantics for activity diagrams. The composition is based on fundamental semantic constructs with options enabling the definition of

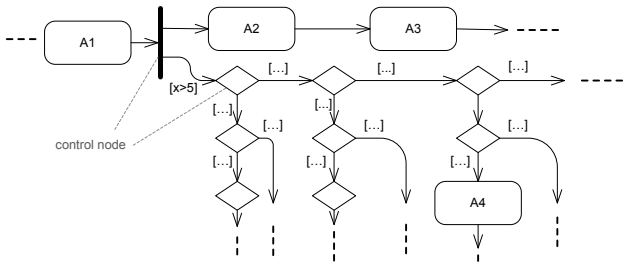

Figure 1: Complex system decisions

domain specific variants. By our approach, a clear and intuitive operational semantics for tool development can be composed which conforms to the informal semantics description of the UML.

As an example, we introduce two tool developments based on particular operational semantics composed in our approach. In both cases, activity diagrams are considered for the specification of system requirements during the early development phase, which is used for validation. The semantics as well as the tools are well evaluated at research projects and used for realistic system developments. The first tool environment focuses on the modeling of information system behavior [Deynet et al., 2010] whereas the second tool aims at the modeling of reactive system behavior [Knieke and Goltz, 2010].

2 Abstract Syntax of UML 2 Activity Diagrams

Activity diagrams are described by a directed graph [OMG, 2009]. Activity edges can be labeled by guards. There are different kinds of action nodes (see Fig. 2): Activities may invoke other activities using call behavior actions where the call of an activity is indicated by placing a rake-style symbol within the node. Accept event actions are actions waiting for an event occurrence triggered by send signal actions. UML provides seven kinds of control nodes, e.g. for specifying parallel and optional flows. Flow final nodes destroy all tokens that arrive at them while activity final nodes terminate all tokens in an activity. An important improvement is the introduction of interruptible activity regions for terminating the token flow in a portion of an activity by traversing interrupting edges.

Definition 1 (Activity Node) The set of activity nodes N of an activity is a union of disjoint sets: $N = N^{act} \dot\cup N^{c}$, where N^{act} is a set of action nodes and N^{c} is a set of control nodes. The set of action nodes N^{act} of an activity is a union of disjoint sets: $N^{act} = N^{ba} \dot\cup N^{cb} \dot\cup N^{ss} \dot\cup N^{ae}$ where N^{ba} is a set of (basic) actions, N^{cb} is a set of call behavior actions, N^{ss} is a set of send signal actions, and N^{ae} is a set of accept event actions. We introduce the sets ACT of action expressions, and Σ of events for the labeling of action nodes. The set of control nodes N^{c} of an activity is a union of disjoint sets: $N^{c} = N^{init} \dot\cup N^{afinal} \dot\cup N^{ffinal} \dot\cup N^{dec} \dot\cup N^{merge} \dot\cup N^{fork} \dot\cup N^{join}$, where N^{init} is a set of initial nodes, N^{afinal} is a set of activity final nodes, N^{ffinal} is a set of flow final nodes, N^{dec} is a set of decision nodes, N^{merge} is a set of merge nodes, N^{fork} is a set of fork nodes, and N^{join} is a set of join nodes.

Definition 2 (Activity) An activity $\alpha \in \mathcal{A}$ is a tuple[1] $\langle N, E, IAR,$ source, target, action, guard, event, iar, interr $\rangle$, where N denotes a set of activity nodes and E a set of activity edges. IAR denotes a set of assigned interruptible activity regions. source : $E \rightarrow N$ gives the source node of an activity edge, and target : $E \rightarrow N$ gives the target node of an activity edge. The function action : $N^{ba} \rightarrow ACT$ gives an action expression $a \in ACT$ as the labeling of an action. An activity edge may have a guard. The guard defines a condition for the control flow. guard : $E \rightharpoonup B$ gives a boolean expression $b \in B$ as the guard on an edge. event : $N^{ss} \cup N^{ae} \rightarrow \Sigma$ is a function for the labeling of a send signal action/accept event action by an event $\sigma \in \Sigma$. The partial function iar : $N \cup IAR \rightharpoonup IAR$ returns for an activity node the interruptible activity region by which it is directly surrounded. The function iar is also applied on interruptible activity regions to support the modeling of nested regions. interr : $E \rightharpoonup IAR$ is a partial function, which maps an interrupting edge to the assigned interruptible activity region. The source node of an interrupting edge must be in the region that is interrupted by the edge.

Definition 3 (Activity Diagram Specification) An activity diagram specification AS is defined as a pair: AS = $\langle \mathcal{A}, p \rangle$ where $\mathcal{A}$ is a finite set of activities, and p is a function $p : N^{cb} \rightarrow \mathcal{A}$ from call behavior actions into activities of $\mathcal{A}$.

[1]Notational conventions: Symbol $\rightarrow$ denotes a function, symbol $\rightharpoonup$ denotes a partial function, symbol $\leftrightarrow$ denotes a bijective function, and $\Rightarrow$ denotes an implication.

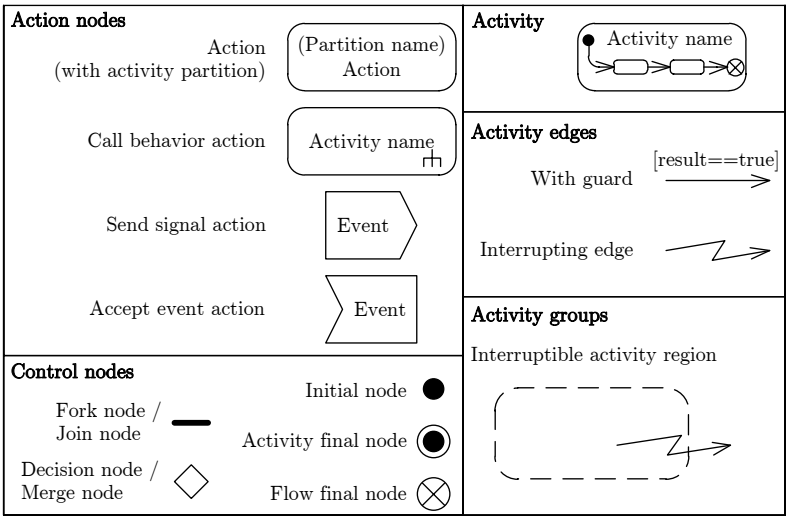

Figure 2: Graphical elements of activity diagrams

3 Fundamental Semantic Constructs

The semantics of activity diagrams is explained in terms of token flow rules inspired by Petri nets [OMG, 2009]. Tokens of the UML activity specification are located on elements of the abstract syntax. We use the term *location* to describe places where tokens can reside. If a token is placed on a location, it is called *active location*. Depending on the domain, locations may be mapped to different positions.

Definition 4 (Locations) For an activity α, let L be the set of locations in α. The assignment of a location to elements of the abstract syntax is done by a domain-dependent bijective function loc (cf. Section 4).

Definition 5 (Enabled Edge) An activity edge $e \in E$ may be enabled, (i.e. e can be traversed), if all assigned locations are active (according to the concept traverse to completion). Depending on the domain, an edge may be realized as *compound edge*. A compound edge consists of a set of edges connected by control nodes.

Definition 6 (Enabled Node) An activity node $n \in N$ may be enabled, (i.e. n can be executed), if relevant e $\in$ E with $n = target(e)$ are enabled.

3.1 Activity Invocation and Status

UML activities may be invoked by behavioral features (e.g. methods and messages) or directly by other activities. During execution an activity has a specific state. This state is defined among others by the active locations. Concurrent execution may lead to multiple activations of a location at the same time. Thus, we introduce a multiset M of active locations for describing the state of an executed activity.

An activity may be invoked more than once concurrently. In our approach the state of invoked activities is described by the term *activity invocation* (AI). At execution of a call behavior action a subordinate AI is executed. One AI may have more than one subordinate activity invocation. This is the case at parallel execution. As depicted in Fig. 3 AIs and their relationships form a *hierarchy*. When the execution of an activity is finished the control flow returns to the caller.[2] An activity may call itself recursively. At this, activities occur on various layers of the hierarchy. These prerequisites result in the following semantic construct:

[2] For lack of space, we give a semantics for synchronous call behavior actions only.

Definition 7 (Activity Invocation) AI is defined by a tuple $\langle i, c, F, M \rangle$, where i is a globally unique identifier of the activity invocation, c defines the reference back to the caller of an activity invocation and may for instance point to an action or an assigned location, F is the set of current subordinate activity invocations denoted by the corresponding invocation identifiers, and M is the multiset of active locations of α.

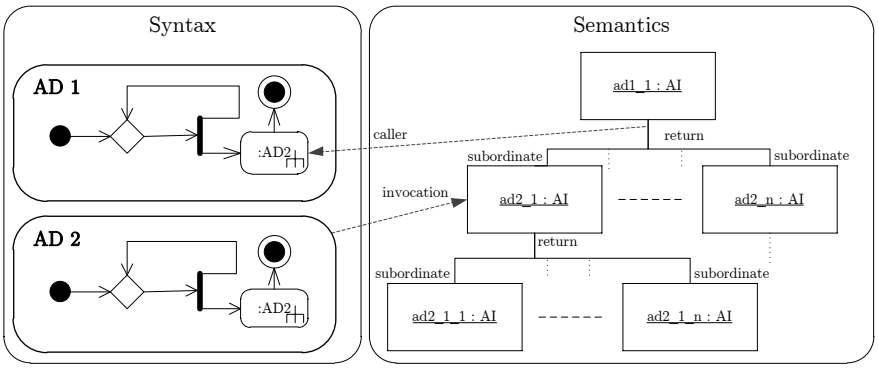

Figure 3: Hierarchy of activity invocations

Definition 8 (Status) A status Π for an activity diagram specification AS is defined as a set of activity invocations. For instance, in Fig. 3 all depicted AIs are part of the status.

Call behavior actions may lead to new AIs within a status, whereas final nodes may remove AIs from a status.

Definition 9 (Initial Locations) At the beginning of the execution of an activity invocation AI, certain locations are entered in AI. The set of so called initial locations of an activity α is denoted as $\overline{M}_\alpha$.

3.2 Internal and External Steps following a Global Clock

In order to provide a well defined execution sequence of UML activities it is crucial to synchronize the processing of active locations. In our approach this synchronization is realized by steps following a global clock. At a step every active location is processed once. A step consists of an internal and an external step. At the internal step active locations are moved from a node to the outgoing edges. During the external step active locations are moved from edges to the next node. Depending on the domain different realizations of the UML activity semantics are supported. For instance, an invocation of an activity may be initialized at the external or at the internal step.

3.3 Step Algorithm

The execution of an activity model is described by a step algorithm. A step is divided into two phases: Leaving nodes (internal step) and traversing enabled edges (external step). The main realization of the global clock with an external and internal step is described by algorithm 1. The order of the internal and external step is undeterminded.

Algorithm 1 Global clock with an external and internal step phase

```
1: while TRUE do
2:     Step preparation
3:     Compute the contents of the internal step
4:     Execute the internal step
5:     Compute the contents of the external step
6:     Execute the external step
7: end while
```

3.4 Interruptible Activity Regions

An important concept of UML activities supported by our approach is the concept of *interruptible activity regions* (IAR). At the interruption of an IAR by an interrupting edge, all included active locations are removed from the multiset M of the associated activity invocation. Additionally all subordinate activity invocations invoked by an included call behavior action are removed from the status. IARs are allowed to be nested and several interrupting edges may be passed concurrently. In this case it is crucial to prioritize the handling of interrupting edges. In our approach this is done by their nesting. Inner IARs have less priority.

Algorithm 2 describes the computation of the external step. The computation is defined by the handling of enabled interrupting edges.

Algorithm 2 Compute the contents of the external step

```
1: Compute the multiset EE of enabled edges
2: Compute the multiset IE of enabled interrupting edges
3: Remove all enabled interrupting edges, for which enabled interrupting edges
   with a higher priority exist, from IE
4: if IE is not empty then
5:     IE constitutes the step
6: else if EE is not empty then
7:     EE constitutes the step
8: else
9:     external step is empty
10: end if
```

The realization of the step preparation and execution depends on the semantic options assigned to locations, activity invocations and the domain specific definition of the external and internal step.

4 Domain Specific Semantics

4.1 Operational Semantics for Specifying Information Systems

At the validation of activity models for information systems the user of the simulator may want to be involved at system decisions. To enable a tracking of the current decision locations have to be mapped to control nodes. The following semantic options of the fundamental semantic constructs described in Section 3 define the operational semantics for the information system domain:

Mapping Variant Loc. Let L be a set of locations in activity α. The bijective function $\mathsf{loc} : \{\langle e, \mathsf{target}(e)\rangle \mid e \in E\} \dot\cup \{\langle e, \mathsf{source}(e)\rangle \mid e \in E\} \leftrightarrow L$ assigns the locations to the begin and the end of activity edges. Fig. 4 depicts an example action with locations at the associated edges.

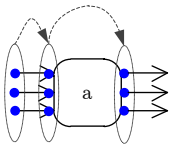

Figure 4: Location mapping variant for information systems

Mapping Variant Initial Locations. $\overline{M}_\alpha$ of the activity invocation contains all locations corresponding to the outgoing edges of an initial node.

Mapping Variant Enabled Edge. An edge $e_i \in E$ in an activity invocation $\langle i, c, F, M\rangle$ is enabled, iff the location $\mathsf{loc}(\langle e_i, \mathsf{source}(e_i)\rangle)$ is at least once in M.

Mapping Variant Enabled Node. A node $n_i \in N$ in an activity invocation $\langle i, c, F, M\rangle$ which is not a merge node is enabled, iff $\mathsf{loc}(\langle e, n_i\rangle) \in M$ for all $e \in E$ with $n_i = \mathsf{target}(e)$. A merge node is enabled, iff $\mathsf{loc}(\langle e, n_i\rangle) \in M$ for at least one $e \in E$ with $n_i = \mathsf{target}(e)$.

Step Algorithms. Algorithms 3, 4 and 5 describe the domain specific step algorithms. At Fig. 4 the external and the internal step are shown by an example. At the external step exactly one location is moved from $\mathsf{loc}(\langle e, \mathsf{source}(e)\rangle)$ to $\mathsf{loc}(\langle e, \mathsf{target}(e)\rangle)$. At the internal step one location of every incoming edge of $a = \mathsf{target}(e)$ is removed and a is executed. After execution, one location of every outgoing edge e' of a is added to M.

Algorithm 3 Execute the external step

1: **for all** enabled edges e_i and $j = \langle i, c, F, M\rangle$ of the external step **do**
2: $M \leftarrow M \ominus \mathsf{loc}(\langle e_i, \mathsf{source}(e_i)\rangle)$ (*multiset removal*)
3: $M \leftarrow M \uplus \mathsf{loc}(\langle e_i, \mathsf{target}(e_i)\rangle)$ (*multiset sum*)
4: **if** e_i is an enabled interrupting edge belonging to an iar $\in$ IAR **then**
5: delete from M all locations situated in iar, and terminate all subordinate synchronous activity invocations
6: **end if**
7: **end for**

Activity Node Execution. The behavior of an activity node depends on the kind of the node. In the following activity node executions are described: A list of several basic action kinds and their behavior are described in [Deynet et al., 2010]. Generally the execution of an activity node is followed by algorithm 5, which adds tokens to the outgoing edges.

Let $n_i \in EN$ be the executed synchronous call behavior action and $j = \langle i, c, F, M\rangle$ be the assigned activity invocation. At the execution a new AI is added to the status $\Pi \leftarrow \Pi \cup \{\langle i', n_i, \emptyset, \overline{M}_{p(n_i)}\rangle\}$ and i' is added to the set of subordinate activity invocations $F \leftarrow F \cup \{i'\}$ of j. In contrast to the synchronous case, an asynchronous call behavior action is finished directly after the AI is added to the status. At the execution of a send signal action the user of the simulator gets a list of all activity invocations and the accept event actions of these invoked activities. The user selects the desired accept event action, which stores the signal in a buffer. An executed accept event action ae waits until at least one signal is stored in its buffer. If this is the case, one signal is removed from the buffer and ae is finished.

At the execution of a decision node a dialog is shown for simulating a system decision (e.g. the user is logged in or not) by a user. The content of the dialog depends on the guards of the outgoing edges. One guard can be accepted by the user. Afterwards the execution is finished. Fork and join nodes behave similar to decision nodes, in constrast the user can accept several guards and edges without guard are accepted by default (see algorithm 5).

Let $j' = \langle i, c, F, M\rangle$ be the activity invocation of the executed activity final node. At the execution of the activity final node j' is removed from the status $\Pi \leftarrow \Pi \setminus \{j'\}$, all subordinate synchronous activity calls and the current step

Algorithm 4 Compute and execute the internal step

1: Compute the set of enabled nodes EN
2: **for all** $n_i \in EN$ and $j = \langle i, c, F, M \rangle$ **do**
3: **if** n_i is *merge node* **then**
4: Let e be one edge with $n_i = \mathsf{target}(e)$ and $\mathsf{loc}(\langle e, n_i \rangle) \in M$
5: $M \leftarrow M \ominus \mathsf{loc}(\langle e, n_i \rangle)$ (*multiset removal*)
6: **else**
7: **for all** $e \in E$ with n_i = target(e) **do**
8: $M \leftarrow M \ominus \mathsf{loc}(\langle e, n_i \rangle)$ (*multiset removal*)
9: **end for**
10: **end if**
11: Start execution of n_i
12: **end for**

are terminated. If $c \neq \emptyset$ the execution of c is finished and j' is removed from the set of subordinate activity calls of F.

Algorithm 5 Finish the execution of a node

1: Let node n_i be the node whose execution is finished
2: Let $\langle i, c, F, M \rangle$ be the activity invocation of the node execution
3: **for all** $e \in E$ with n_i = source(e) **do**
4: **if** e has no guard or guard is accepted by the user **then**
5: $M \leftarrow M \uplus \mathsf{loc}(\langle e, \mathsf{source}(e) \rangle)$ (*multiset sum*)
6: **end if**
7: **end for**

Fig. 5 shows an example activity Y. Let a-g be basic actions and activity invocation $j = \langle 0, \emptyset, \emptyset, \{Y_1, Y_2\} \rangle$ be initialized at the execution of activity Y. For simplification we assume that the execution of all actions take the same time. Applying the step algorithm, M may be adapted corresponding to the following sequence: $\{Y_1, Y_2\}$, $\{Y_3, Y_4\}$, $\{Y_5, Y_6\}$, $\{Y_7, Y_8\}$, $\{Y_9, Y_{10}\}$, $\{Y_{11}$, $Y_{12}\}$, $\{Y_{12}$, $Y_{13}\}$, $\{Y_{12}$, $Y_{14}\}$, $\{Y_{13}$, $Y_{15}, Y_{16}\}$, $\{Y_{14}, Y_{17}$, $Y_{18}\}$, $\{Y_{15}, Y_{16}$, $Y_{19}, Y_{20}\}$, $\{Y_{17}, Y_{18}$, $Y_{21}, Y_{22}\}$, $\{Y_{19}$, Y_{20}, $Y_{23}\}$, $\{Y_{21}, Y_{22}$, $Y_{24}\}$, $\{Y_{23}$, $Y_{25}\}$, $\{Y_{24}$, $Y_{26}\}$. After this sequence j is removed from the status.

4.2 Operational Semantics for Specifying Reactive Systems

Reactive systems are characterized by a steady interaction with the system's environment and are often composed of multiple interacting subsystems. Thus, we enable the user to interfere with the system by inducing events like pushing a button, or operating some input device. Events can be received by accept event actions which may trigger the execution of activities. We have defined an action language which is executed by the execution engine [Knieke and Goltz, 2010].

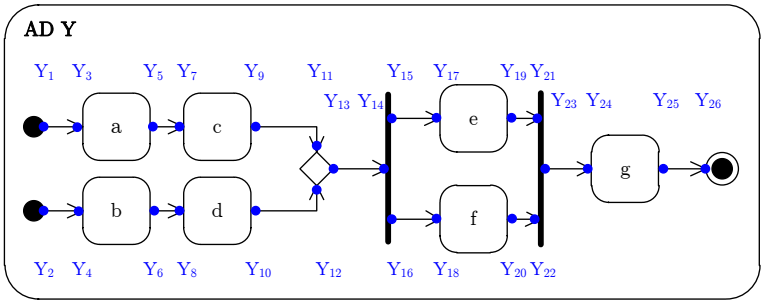

Figure 5: Activity diagram with locations and example workflow

The token flow semantics differs from the semantics for information systems from Section 4.1. Here, action execution is done by the system according to predefined operations. Thus, we place a location on the node were the system may reside in during execution of the action node (see Fig. 6).

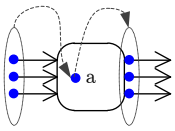

Figure 6: Location mapping variant for reactive systems

Mapping Variant Locations. For an activity α, let L be a set of locations in α. Let $N' = N^{act} \cup N^{init}$ be the union of sets of action nodes, and initial nodes in α and $E' = \{e \in E \mid \mathsf{source}(e) = n, n \in N'\}$ be a set of edges outgoing from nodes of N'. Locations are assigned to nodes and edges by a bijective function $\mathsf{loc} : N' \cup E' \leftrightarrow L$ that gives a location of L for each node of N' and for each edge of E'.

Mapping Variant Initial Locations. $\overline{M}_\alpha$ contains locations of initial nodes and of accept event actions which have no incoming edges, and which are not located in an interruptible activity region.

In contrast to the operational semantics for information systems, the user is not involved in the evaluation of guard conditions at control nodes like decision nodes. Thus, multiple control nodes between two action nodes can be evaluated all at once. Hence, we do not need locations on control nodes. During a step, sets of actions linked by control nodes are traversed which we call compound edges. A compound edge (CE) is a set of edges that are linked by AND (fork/join) and OR (decision/merge) nodes.

- If an edge in a compound edge enters or leaves an AND node, then every edge that leaves or enters the AND node is part of the compound edge.
- If an edge in a compound edge enters (leaves) an OR node, then there is exactly one edge in the compound edge that leaves (enters) the OR node.

Mapping Variant Enabled Edge. Let AS be an activity diagram specification and Π a status of AS. A CE in an activity invocation $\langle i, c, F, M\rangle$ (denoted by ce_i) is enabled, iff its locations are at least once in M and the guard is evaluated to TRUE.

The set of locations associated with the edges of ce is denoted by ${}^\bullet ce$ and $ce^\bullet$ denotes the set of locations entered after ce has been traversed.

Step Algorithms. Next, we give a domain specific realization of the phases of algorithm 1 (p. 7). As the computation of the external step is generally defined in algorithm 2, we define the phases step preparation, computation/execution of internal step, and execution of the external step by algorithms. We have adopted some general principles from [Harel and Naamad, 1996] in defining a step, e.g. reactions to event occurrences can be sensed only after completion of the step, and events live for the duration of one step only, the one following that in which they occur [Knieke and Goltz, 2010]. At the step preparation phase, external event occurrences are added to the list of internally generated events and values of data-items implied by external changes are adjusted.

Actions that have finished execution since the last step, as well as initial and object nodes entered in the last step are left by executing the internal step.

Algorithm 6 Compute and execute the internal step

1: **for all** $\langle i, c, F, M\rangle \in \Pi$ **do**
2: Let $S \subseteq M$ be a multiset of locations l fulfilling the following prerequisites:
3: l is assigned to a node n
4: n is *action node* $\Rightarrow$ action execution has finished
5: n is *call behavior action node (synch.)* $\Rightarrow$ invocation has returned
6: n is *accept event action node* $\Rightarrow$ appropriate event occurred
7: **for all** $s \in S$ **do**
8: Update the variables affected by action execution
9: Remove s from M
10: Add all locations l' of the outgoing edges of the node of s to M
11: **end for**
12: **end for**

Algorithm 7 Execute the external step

1: **for all** CEs ce_i and $j = \langle i, c, F, M \rangle$ of the external step **do**
2: $M \leftarrow M \ominus {}^\bullet ce_i$ (*multiset removal*)
3: **if** IAR is interrupted by *enabled interrupting CE* **then**
4: delete from M all locations situated in the IAR, and terminate all subordinate synch. activity invocations
5: **end if**
6: **if** a target node of ce_i is an *activity final node* **then**
7: $\Pi \leftarrow \Pi \setminus \{j\}$, terminate all subordinate synchronous activity calls, and terminate the step
8: **if** $c \neq \emptyset$ **then**
9: return to the calling activity invocation
10: **end if**
11: **end if**
12: $M \leftarrow M \uplus ce_i^\bullet$ (*multiset sum*)
13: **if** *IAR* is enabled in j **then**
14: accept event actions that that do not have incoming edges are enabled
15: **end if**
16: Start execution of entered action nodes
17: **if** *synchronous call behavior action* t entered in j **then**
18: $\Pi \leftarrow \Pi \cup \{\langle i', \mathsf{loc}(t), \emptyset, \overline{M}_{p(t)} \rangle\}$
19: $F \leftarrow F \cup \{i'\}$
20: **end if**
21: **if** *send signal action* entered in j **then**
22: generate internal event occurrence for the next step
23: **end if**
24: **end for**

Fig. 7 shows an example activity Y. Let a-g be basic actions and activity invocation $j = \langle 0, \emptyset, \emptyset, \{Y_1, Y_2\}\rangle$ be initialized at the execution of activity Y. For simplification we assume that the execution of all actions take the same time. Applying the step algorithm, M may be adapted corresponding to the following sequence: $\{Y_1, Y_2\}$, $\{Y_3, Y_4\}$, $\{Y_5, Y_6\}$, $\{Y_7, Y_8\}$, $\{Y_9, Y_{10}\}$, $\{Y_{11}, Y_{12}\}$, $\{Y_{13}, Y_{13}, Y_{14}, Y_{14}\}$, $\{Y_{15}, Y_{15}, Y_{16}, Y_{16}\}$, $\{Y_{17}, Y_{17}\}$, $\{Y_{18}, Y_{18}\}$. After this sequence j is removed from the status.

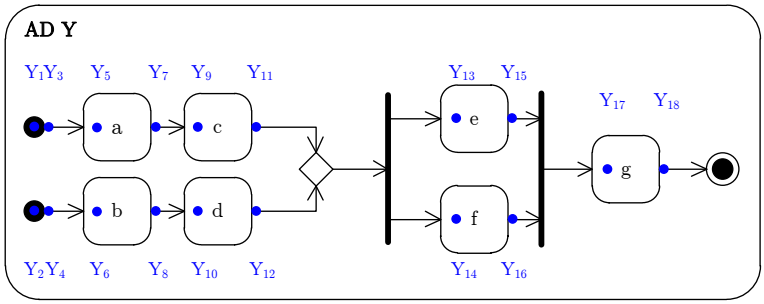

Figure 7: Activity diagram with locations and example workflow

5 Related Work

[Eshuis, 2002] proposes the use of UML 1.x activity diagrams for workflow modeling. Eshuis presents an operational semantics influenced by statecharts semantics and provides tool support for verification. However, Eshuis refers to UML 1.x and omits a number of interesting elements including hierarchy. Eshuis regards actions as a kind of external behavior executed by actors.

[Störrle and Hausmann, 2005] investigates the mapping of UML 2 activity diagrams to Petri nets as the UML standards 2.x explicitly refer to Petri nets when explaining the semantics of activity diagrams. Störrle shows that a homogeneous transformation of the rich set of concepts of activity diagrams to an established Petri net variant is far from trivial. In [Störrle and Hausmann, 2005], it is shown that concepts like call behavior actions or interruptible regions cannot be handled in the same way as basic actions and simple control nodes when transforming them to Petri nets.

In an alternative approach, [Sarstedt, 2006] define the token flow semantics of UML 2 activity diagrams using Abstract State Machines (ASM) (cf. [Börger and Stärk, 2003]). Sarstedt's formal definition covers most elements of UML 2 activity diagrams dealing with control and data flow, signal handling, nested interruptible activity regions, and buffering. The ASM formalization serves as a semantic basis for a simulation environment for embed-

ded system design: [Sarstedt, 2006] models the structure of embedded applications by UML 2 class diagrams. The behavior of a class is specified in an associated *ActiveChart*.

[Engels et al., 2007, Engels et al., 2009] use dynamic meta modeling to describe the behavioral semantics of UML activity diagrams. Dynamic meta modeling extends the metamodel defining the syntax of a modeling notation [Soltenborn and Engels, 2009]. The extended metamodel – the so-called *runtime metamodel* – provides concepts for describing states (configurations) of an executed model: The configurations are described by the models itself extended by information on the currently active elements. The graph transformation rules specify how to transit to the next configuration by modifying instances of the runtime metamodel. Thus, the graph transformation rules correspond to the step algorithm defined in the previous section.

In [Staines, 2010] a formal mapping from activity diagrams to Petri nets based on Triple Graph Grammar (TGG) rules is defined as an extension of Störrles approach. Unlike Störrles simple mapping, TGG rules constitute a model-to-model transformation. In addition, the transformation is executable and allows for the construction of complex rules for supporting semantically intricate constructs. [Staines, 2010] define the mapping for simple place/transition nets. For applying the TGG approach to the complete language elements of activity diagrams, additional classes of Petri nets have to be taken into account. However, it is still an open question, whether a unique class of Petri nets exists that can serve as a foundation for the transformation supporting all elements of UML 2 activity diagrams and for which analysis tools are available.

[Grönniger et al., 2010] define a formal semantics for a subset of activity diagrams which allows for a domain-specific interpretation of activity diagrams: variants determine which system entities make up a diagram instance. They focus on rather low-level interpretations of activity diagrams as simple action or method executions. Advanced constructs like interruptible activity regions are currently not supported. The semantics is encoded in a theorem prover as a basis for verification [Grönniger et al., 2010, p. 5]. In contrast, our algorithmic definitions of the semantics aim at tool development for fully executable models. [Grönniger et al., 2010] do not handle conflicts/interferences which may arise if multiple instances of activities are executed concurrently [Grönniger et al., 2010, p. 8]. Furthermore, their variation points concerning the token flow semantics are not able to support the concept of compound edges w.r.t. complex concatenations of control nodes (e.g. decision and fork) which will not be traversed instantaneously according to [Grönniger et al., 2010].

Finally, we mention approaches defining a composable semantics for model-based notations (e.g. [Niu et al., 2002]). In [Esmaeilsabzali and Day, 2010] a formal framework to define the semantics of *Big-Step Modelling Languages*

(BSMLs) is described. A BSML is a language in which a model can respond to an input of the environment via a sequence of *small steps*, each of which may consist of the concurrent execution of a set of transitions [Esmaeilsabzali and Day, 2010]. The semantics of many BSMLs can be deconstructed into eight high-level semantic aspects and their semantic options. As an example, a set of semantic options of the STATEMATE-semantics of statecharts is given [Esmaeilsabzali and Day, 2010]. The semantic framework for deriving a formal semantics of a BSML defined in [Esmaeilsabzali and Day, 2010] contains a set of parameters. By adjusting these parameters, an executable BSML-semantics can be derived.

6 Conclusion

In this paper, we analyzed common semantical constructs and possible options for the definition of operational semantics for activity diagrams. At this, we introduced a foundation for a framework, which uses these fundamental semantic constructs for the composition of operational semantics. The provided options of these contructs go beyond the capabilities of the UML superstructure. Two examples demonstrate the applicability of the approach. For the information system domain locations at control nodes enable an involvement of the user in system decisions. The semantics for the reactive systems domain enable, for instance, an exact concurrent execution by the definition of compound edges. Both semantics supports new UML 2.0 features like hierarchy and signals. This paper focuses on control flow of activity diagrams. As a future work common semantic constructs for the description of data flow will be considered in detail.

References

[Börger and Stärk, 2003] Börger, E. and Stärk, R. (2003). *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag.

[Deynet et al., 2010] Deynet, M., Niebuhr, S., Rausch, A., and Schindler, B. (2010). Enhancing Validation with Prototypes out of Requirements Models. In *21st Australian Software Engineering Conference (ASWEC 2010 Industry Track)*.

[Engels et al., 2009] Engels, G., Fisseler, D., and Soltenborn, C. (2009). Improving Reusability of Dynamic Meta Modeling Specifications with Rule Overriding. In *Proc. of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2009*, pages 39–46.

[Engels et al., 2007] Engels, G., Soltenborn, C., and Wehrheim, H. (2007). Analysis of UML Activities Using Dynamic Meta Modeling. In *Proc. of the 9th IFIP WG 6.1 Intern. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2007)*, volume 4468 of *LNCS*, pages 76–90. Springer-Verlag.

[Eshuis, 2002] Eshuis, H. (2002). *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, CTIT, University of Twente.

[Esmaeilsabzali and Day, 2010] Esmaeilsabzali, S. and Day, N. A. (2010). Prescriptive Semantics for Big-Step Modelling Languages. In *Proc. of the 13th Intern. Conf. on Fundamental Approaches to Software Engineering (FASE 2010)*, volume 6013 of *LNCS*, pages 158–172. Springer-Verlag.

[Grönniger et al., 2010] Grönniger, H., Reiss, D., and Rumpe, B. (2010). Towards a Semantics of Activity Diagrams with Semantic Variation Points. In Petriu, D. C., Rouquette, N., and Haugen, Ø., editors, *MODELS 2010*, volume 6394 of *LNCS*, pages 331–345. Springer.

[Harel and Naamad, 1996] Harel, D. and Naamad, A. (1996). The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodologies*, 5(4):293–333.

[Knieke and Goltz, 2010] Knieke, C. and Goltz, U. (2010). An Executable Semantics for UML 2 Activity Diagrams. In *ECOOP 2010 Workshop Proc. of the Intern. Workshop on Formalization of Modeling Languages (FML 2010)*, pages 11–15. ACM Press.

[Niu et al., 2002] Niu, J., Atlee, J. M., and Day, N. A. (2002). Composable Semantics for Model-based Notations. *SIGSOFT Softw. Eng. Notes*, 27:149–158.

[OMG, 2009] OMG (2009). UML, Version 2.2. OMG Specification Superstructure and Infrastructure, `http://www.omg.org`.

[Sarstedt, 2006] Sarstedt, S. (2006). *Semantic Foundation and Tool Support for Model-Driven Development with UML 2 Activity Diagrams*. PhD thesis, Universität Ulm.

[Soltenborn and Engels, 2009] Soltenborn, C. and Engels, G. (2009). Towards Test-Driven Semantics Specification. In *Proc. of the 12th Intern. Conf. on Model Driven Engineering Languages and Systems (MODELS 2009)*, volume 5795 of *LNCS*, pages 378–392. Springer-Verlag.

[Staines, 2010] Staines, A. (2010). A Triple Graph Grammar Mapping of UML 2 Activities into Petri Nets. *International Journal of Computers*, 4(1):27–35.

[Störrle and Hausmann, 2005] Störrle, H. and Hausmann, J. H. (2005). Towards a Formal Semantics of UML 2.0 Activities. In *Software Eng. 2005, Fachtagung des GI-Fachbereichs Softwaretechnik*, volume 64 of *LNI*, pages 117–128. Springer-Verlag.